Análisis de Caso: Plataforma Web Institucional Escalable con Contenedores

1. Contexto y Justificación

La Universidad Latina de Costa Rica está modernizando su infraestructura para hospedar sitios web institucionales con tecnologías como Java, NodeJS, Drupal y WordPress. El objetivo es garantizar alta disponibilidad, seguridad y escalabilidad mediante el uso de contenedores Docker, balanceo de carga con HAProxy y automatización CI/CD. Esta arquitectura optimiza recursos, mejora la resiliencia y permite responder eficientemente a picos de demanda.

2. Objetivo del Proyecto

Diseñar y simular una plataforma web institucional que cumpla con los principios de escalabilidad, alta disponibilidad y seguridad, mediante contenerización de servicios, balanceo de carga, simulación de fallos, implementación de HTTPS y análisis de rendimiento.

3. Desarrollo Técnico y Análisis

3.1. Simulación de un Servicio en Contenedor

Se crea un Dockerfile con la imagen base openjdk:17, se copia el archivo .jar y se expone el puerto 8080. Se ejecuta el contenedor localmente y se evalúa el aislamiento de procesos, red y persistencia.

Ejemplo de Dockerfile:
FROM openjdk:17
COPY app.jar /app.jar
EXPOSE 8080
CMD ["java", "-jar", "/app.jar"]

3.2. Balanceo de Carga con HAProxy

Se levantan dos instancias del contenedor Java y se configura HAProxy para distribuir el tráfico. Se utiliza balanceo round robin y se verifica con navegador o curl.

3.3. Alta Disponibilidad

Se simula la caída de una instancia para verificar que HAProxy mantenga el servicio activo. Opcionalmente se utiliza Docker Swarm para replicación automática.

3.4. Seguridad

Se implementa HTTPS con certificado auto-firmado o Let's Encrypt, se analizan riesgos de HTTP plano y se aplican medidas adicionales como firewalls y control de acceso por roles.

3.5. Escalabilidad y Mantenimiento

Se comparan características con sitios reales, se plantea escalabilidad horizontal con Docker Swarm y se usan volúmenes NFS para persistencia.

4. Resultados Esperados

Plataforma capaz de manejar tráfico distribuido, con recuperación automática ante fallos, conexiones seguras y preparación para escalar horizontalmente.

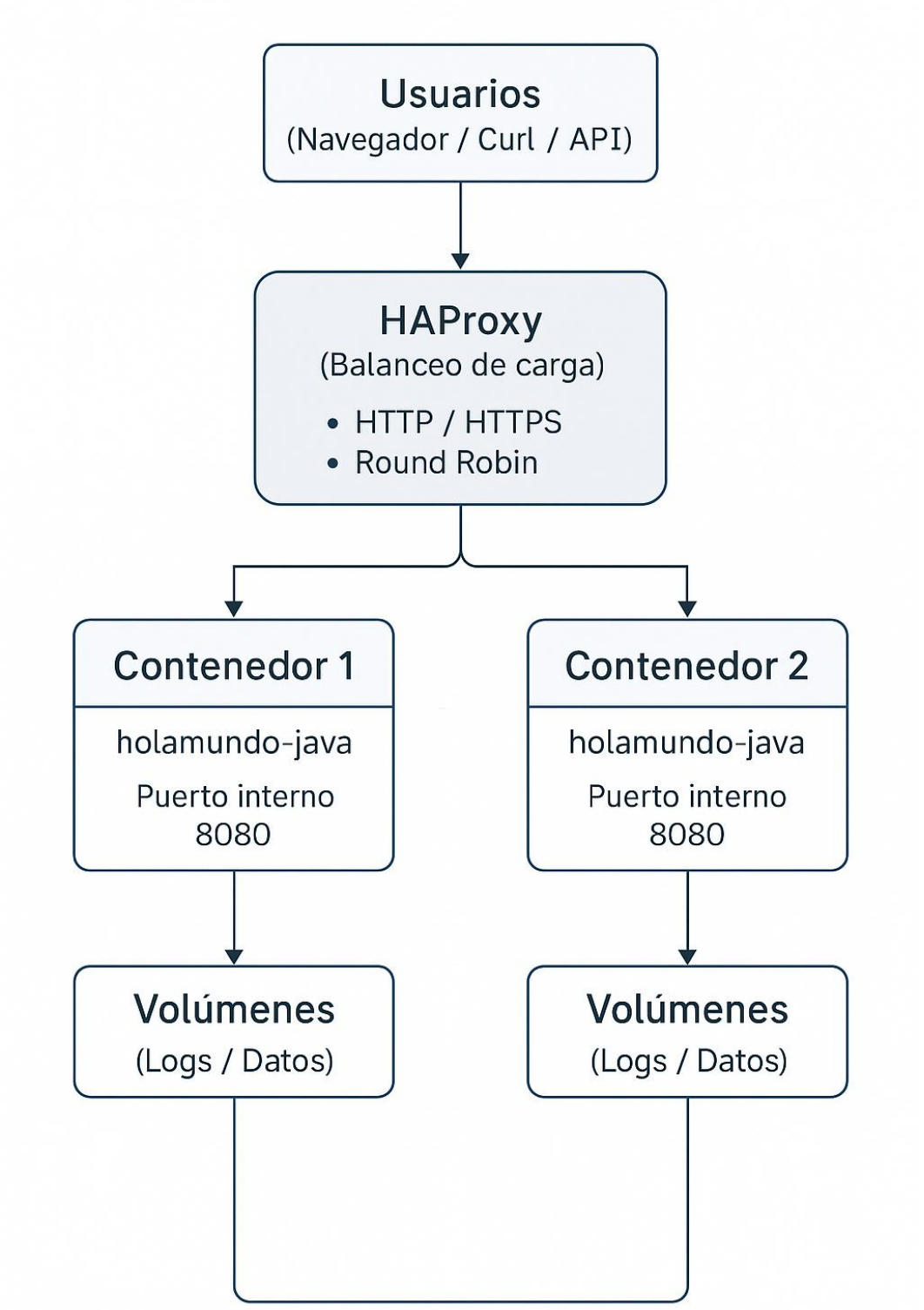
5. Objetivos de Aprendizaje Alcanzados

- Comprender la gestión de puertos en entornos de contenedores.
- Implementar balanceadores de carga en arquitecturas distribuidas.
- Manejar seguridad en aplicaciones web modernas.
- Realizar monitoreo y pruebas de rendimiento.

6. Entregables

- Informe técnico con configuraciones y resultados.
- Capturas de pruebas.
- Diagramas de arquitectura.
- Archivos de configuración (Dockerfile, haproxy.cfg, docker-compose.yml).

7. Diagrama de Arquitectura



8. Archivos de Configuración

docker-compose.yml

```
version: "3.8"
services:
 haproxy:
   image: haproxy:2.7
   container_name: haproxy
   ports:
    - "80:80"
    - "443:443"
   volumes:
    - ./haproxy.cfg:/usr/local/etc/haproxy/haproxy.cfg:ro
    - ./certs:/etc/haproxy/certs:ro
   depends_on:
    - app1
    - app2

 app1:
   build: ./app
   container_name: app1
   ports:
    - "8081:8080"

 app2:
   build: ./app
   container_name: app2
   ports:
    - "8082:8080"

networks:
 default:
   driver: bridge
```

haproxy.cfg

```
global
   log stdout format raw local0

defaults
   log     global
   option  httplog
   option  dontlognull
   timeout connect 5000
   timeout client  50000
```

```
    timeout server  50000

frontend http_front
    bind *:80
    default_backend http_back

frontend https_front
    bind *:443 ssl crt /etc/haproxy/certs/cert.pem
    default_backend http_back

backend http_back
    balance roundrobin
    server app1 app1:8080 check
    server app2 app2:8080 check
```

Dockerfile para la App

```
FROM openjdk:17
COPY app.jar /app.jar
EXPOSE 8080
CMD ["java", "-jar", "/app.jar"]
```